Unit-2

Stack

#Stack:

The stack is an ordered collection of homogeneous data elements where the insertion and deletion operation occur at only one end. This end is usually known as top of the stack. Here, the last element inserted will be top of the stack. Since, deletion is done from the same end, last element inserted will be the first element to be removed out from stack and so on. That is why stack is called LIFO (Last in First Out). Top is incremented while pushing (inserting) an element into the stack and decremented while popping (deleting) an element from the stack. It is named stack as it behaves like a real-world stack, for example - a deck of cards or a pile of plates, etc.

A common model of the stack is placed on the marriage party, fresh plates are "pushed" (inserted) on to the top and "pop" (deleted) from the stack.

If there is no enough space then stack is said to be in over-flow state and no new element can be pushed. Similarly, before pop operation, if stack is empty and pop operation is attempted, the stack is said to be in ~~over~~ underflow state.

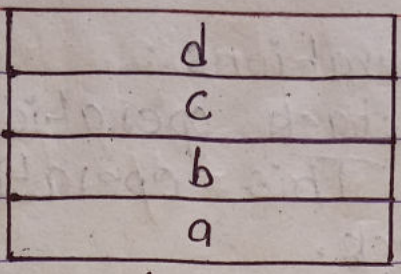


Fig: Stack containing element

Stack Operation:

The basic or primitive operation performed on the stack are:

(a) "PUSH" Operation:

It is used to add (push or insert) elements in a stack. When we read an item to a stack, we say that we pushed it onto the stack. The last item pushed into the stack is at the top.

(b) "POP" Operation:

The pop operation is used to remove or delete top element from the stack. When we remove an item we say that we poped it from the stack. When an item is poped, it is always the top item which is removed.

Some other Operations:

(i) Create-empty stack operation:
This operation is used to create an empty stack.

(ii) IS-full operation:
IS-full operation is used to check whether the stack is full or not. i.e. stack overflow.

(iii) IS-empty operation:
IS-empty operation is used to check whether the stack is empty or not. i.e. stack underflow.

(iv) Top Operation:
Return the elements at the top of the stack.

Stack as an ADT:
A stack of elements of type T is a finite sequence of elements of T together with the operations:

(1) Create Empty Stack (S):
Create or make stack S be an empty stack.

(2) Push (S, x):

Insert x at one end of the stack, called its top.

(3) Top (S):

If stack S is not empty; then retrieve the element at its top.

(4) Pop (S):

If stack S is not empty; then delete the element at its top.

(5) Is Full (S):

Determine whether stack S is full or not. Return true if S is full; return false otherwise.

(6) Is Empty (S):

Determine whether stack S is empty or not. Return true if S is an empty stack; return false otherwise.

Various stages of stacks:

Maximum size = '4' **Push**

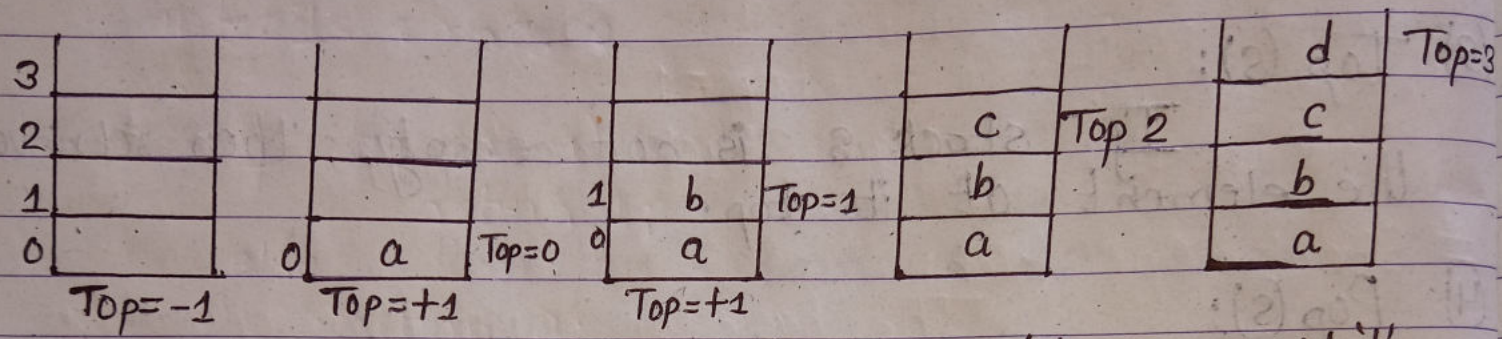
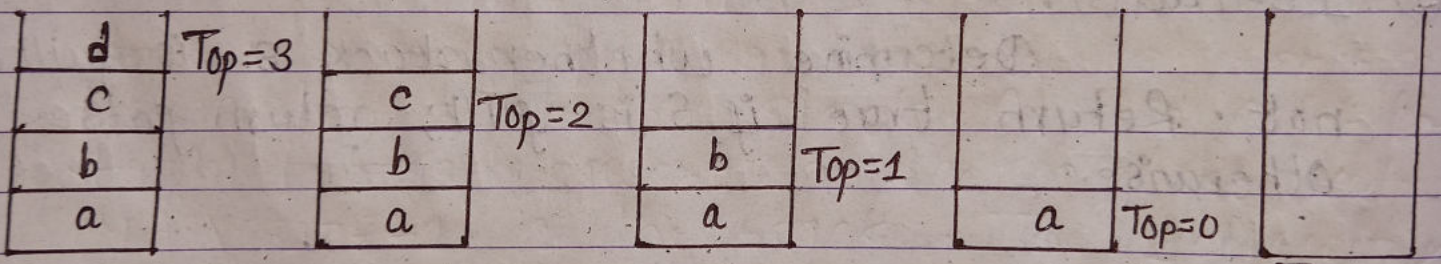


Fig: Empty Stack Insert 'a' Insert 'b' Insert 'c' Insert 'd'

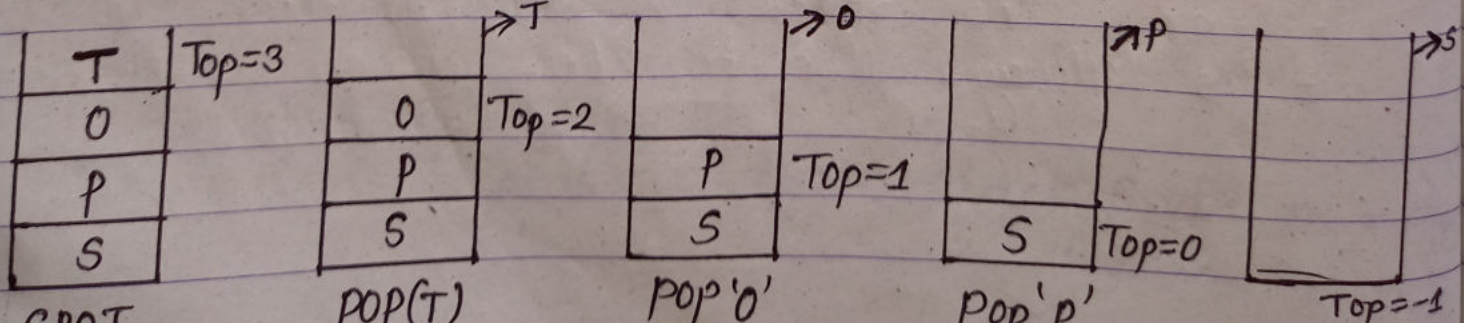
Maximum size = '4' **Pop**



delete 'd' delete 'c' delete 'b' delete 'a'

Application of Stack:

(a) Reversal of a string Reverse = TOPS



SPOT POP(T) POP 'O' Pop 'P' Pop 'S'

(b) To evaluate the expressions (postfix, prefix).
(c) To keep the page-visited history in a web browser.
(d) To perform the undo sequence in a text editor.
(e) Used in recursion.
(f) To check the correctness of parentheses sequence.
(g) To pass the parameter between the functions in a C program.
(h) Tree Traversals.

Creating Empty Stack:
The value of top=-1 indicates the empty stack in C implementation.

```
void create_empty_stack(struct stacke) /* function to
create an empty stack */
{
    e.top=-1;
}
```

Stack Empty or Underflow:
This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
int Is Empty ()
{
    if (top == -1)
        return 1;
    else
        return 0;
}
```

Stack Full or overflow:

This is the situation when the stack becomes full, and no more, elements can be pushed onto the stack. At this point the stack top is present at the highest location (MAX SIZE-1) of the stack. The following function returns true(1) if stack is full, false (0) otherwise.

```
int Is full ()
{
    if (top == MAXSIZE-1)
        return 1;
    else
        return 0;
}
```

Imp

Algorithm for PUSH operation of a stack:

Let stack[MAXSIZE] is an array to implement the stack. The variable top denotes the top of the stack. This algorithm adds or inserts an item at the top of the stack.

1. Start
2. Check for stack overflow as
   if top == MAXSIZE -1 then
   Print "Stack Overflow" and Exit the program
   else
   Increase top by 1 as,
   set, top = top +1
3. Read elements to be inserted say element.
4. Set, stack[top] = element // Inserts item in new position.
5. Stop.

Implementation of push algorithm in C language

```
void push (int val) //n is size of the stack
{
   if (top == MAXSIZE)
        print ("\n Stack Overflow");
   else
     {
        top = top+1;
        stack[top] = val;
     }
}
```

Imp

Algorithm for POP Operation:

This algorithm deletes the top element of the stack and assigns it to a variable element.

1. Start
2. Check for the stack underflow as
   if top < 0 then
   Print "Stack Underflow" and Exit the program.
   else
   Remove the top element and set this element to the variable as
   Set, element = stack[top]
   Decrement top by 1 as
   Set, top = top - 1
3. Print 'element' as a deleted item from the Stack.
4. Stop.

The C function for POP operation:

Alternatively, we can define the POP function as given below:

```
void pop()
{
    int item;
    if (top < 0)  // Checking Stack Underflow
        print(" The Stack is Empty");
```

```
else
  {
    item = stack[top]; // Storing top element to item
                          variable
    top = top - 1  // Decrease top by 1
    printf("The popped item is = %d", item); // Dis-
     playing the deleted items.
  }
}
```

#Visiting each element of the stack (Peek or Display Operation):

Display operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Display algorithm

1. Start
2. Check for the stack underflow as
   if top<0 then
      Print "Stack underflow" and Exit the Program.

   Else
      Set, i = top
      while (i>=0)
         Display Stack[i]
         Decrement i by 1 as
         Set, i = i-1
      End while
3. Stop

Expression:

An expression is defined as the number of operands or data items combined with several operators. An application of stack is calculation of postfix expression. There are basically three types of notation for an expression.

(1) Prefix Expression (Polish):

If the operator symbol are placed before its operands then the expression is called ~~post~~ prefix expression.

(2) Postfix expression (Reverse - Polish):

If the operator symbol are placed after its operand then the expression is called postfix expression.

(3) Infix expression:

If the operator symbol are placed between the operands then the expression is called infix expression.

Operators Precedence:

Exponential Operator	^ Highest precedence (^ = $)
Multiplication / Division	*/' Next Precedence
↓ Addition / Subtraction	+/- least Precedence

Infix to Postfix expression conversion using stack:

Step 1: Scan the expression from left to right
(a) If the symbol is an operand, add it to the postfix expression.
(b) If the symbol is an open paranthesis '(', push it on to the stack.
(c) If the symbol is an operator then add/push it into the top of the stack.

Step 2: If the precedence of the operator at the top of the stack is higher or same as the current (scanned) operator then repeatedly it is poped from stack and added to the postfix expression. Otherwise, it is pushed onto the stack.

Step 3: If the symbol is closing paranthesis then repeatedly pop each operator from the stack and add it to the postfix expression until the corresponding open paranthesis is encountered.

Step 4: Remove the opening paranthesis from stack.

Step 5: Repeat Step 1 to 5 until end of expression.

IMP

Convert the following infix expression to the postfix expression showing stack step after each step in tabular form.

(a) A ^ B * C / (D * E - F)

Symbol Scanned	Stack	Postfix expression
A		A
^	^	A
B	^	AB
*	*	AB^
C	*	AB^C
/	/	AB^C*
(	/(	AB^C*
D	/C	AB^C*D
*	/C*	AB^C*D
E	/C*	AB^C*DE
-	/C-	AB^C*DE*
F	/C-	AB^C*DE*F
)	/	AB^C*DE*F-
	empty	AB^C*DE*F-/

(b) A * (B + D) / E - F * (G + H / K)

Symbol Scanned	Stack	Postfix expression
A		A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
D	*(+	ABD
)	*	ABD+
/	/	ABD+*
E	/	ABD+*E
-	-	ABD+*E/
F	-	ABD+*E/F
*	-*	ABD+*E/F
(	-*(	ABD+*E/F
G	-*(	ABD+*E/FG
+	-*(+	ABD+*E/FG
H	-*(+	ABD+*E/FGH
/	-*(+/	ABD+*E/FGH
K	-*(+/	ABD+*E/FGHK
)	-*	ABD+*E/FGHK/+
	empty	ABD+*E/FGHK/+*-

(c) A+(B*C-(D/E-F)*G)*H

Symbol Scanned	Stack	Postfix expression
A	empty	A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC*
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D
E	+(-(/	ABC*DE
-	+(-(-	ABC*DE/
F	+(-(-	ABC*DE/F
)	+(-	ABC*DE/F-
*	+(-*	ABC*DE/F-
G	+(-*	ABC*DE/F-G
)	+	ABC*DE/F-G*
*	+*	ABC*DE/F-G*-
H	+*	ABC*DE/F-G*-H
	empty	ABC*DE/F-G*-H*+

(d) ((A - (B+C)) * D) $ (E+F)

Symbol Scanned	Stack	Postfix expression
(	(	empty
(	((	empty
A	((	A
-	((-	A
(	((-(	A
B	((-(	AB
+	((-(+	AB
C	((-(+	ABC
)	((-	ABC+
)	(	ABC+-
*	(*	ABC+-
D	(*	ABC+-D
)	empty	ABC+-D*
$	$	ABC+-D*
(	$(	ABC+-D*
E	$(	ABC+-D*E
+	$(+	ABC+-D*E
F	$(+	ABC+-D*EF
)	$	ABC+-D*EF+
	empty	ABC+-D*EF+$

(e) (A+B*C$D)/((G+F-G)*H)$I/J

Symbol Scanned	Stack	Postfix expression
(	(	empty
A	(	A
+	(+	A
B	(+	AB
*	(+*	AB
C	(+*	ABC
$	(+*$	ABC
D	(+*$	ABCD
)	empty	ABCD$*+
~~E~~	~~E~~	~~ABCD$*+~~
~~E~~	~~E~~	~~ABCD$*+~~
~~E~~	~~E~~	~~ABCD$*+E~~
~~E~~	~~E~~	~~ABCD$*+E~~
/	/	ABCD$*+
(	/(	ABCD$*+
(	/((	ABCD$*+
E	/((	ABCD$*+E
+	/((+	ABCD$*+E
F	/((+	ABCD$*+EF
-	/((-	ABCD$*+EF+
G	/((-	ABCD$*+EF+G
)	/(	ABCD$*+EFG-
*	/(*	ABCD$*+EF+G-

Symbol Scanned	Stack	Postfix expression
H	/(*	ABCD$*+EF+G-H
)	/	ABCD$*+EF+G-H*
$	/$	ABCD$*+EF+G-H*
I	/$	ABCD$*+EF+G-H*I
/	/	ABCD$*+EF+G-H*I$/
J	/	ABCD$*+EF+G-H*I$/J
	empty	ABCD$*+EF+G-H*I$/J/

Evaluation of postfix using stack:

To evaluate the expression, we scan the expression from left to right. The steps involved in evaluation ~~from~~ of postfix expression are:-

Step 1: If an operand is encountered, push it onto the stack.

Step 2: If an operator is encountered
a) Pop two elements of stack where A is top element and B is next top element.
b) Evaluate B OP A.
c) Push the result on stack.

Step 3: The evaluated value is equal to the value at the top of the stack.

IMP

9. Evaluate the following postfix expression using stack.

(a) 296+*23*-

Symbol Scanned	Stack	Operation (BOPA)
2	2	
9	2,9	
6	2,9,6	
+	2, 15	A=6, B=9, (9+6=15)
*	~~2, 15~~ 30	A=~~15~~, B=2 (2*15=30)
2	30,2	
3	30,2,3	
*	30,6	A=3, B=2 (2*3=6)
-	24	A=6, B=30 (30-6=24)

(b) 562+*,12,4 /-

Symbol Scanned	Stack	Operation (BOPA)
5	5	
6	5,6	
2	5,6,2	
+	5,8	A=2, B=6 (6+2=8)
*	40	A=8, B=5 (5*8=40)
12	40,12	
4	40,12,4	
/	40,3	A=4, B=12 (12÷4=3)
-	37	A=3, B=40 (40-3=37)

(c) Infix to postfix evaluation:
7+5*3^2/(9-2^2)+6*4

Symbol Scanned	Stack	Postfix expression
7	empty	7
+	+	7
5	+	75
*	+*	75
3	+*	753
^	+*^	753
2	+*^	7532
/	+/	7532^*
(	+/(	7532^*
9	+/(	7532^*9
-	+/(-	7532^*9
2	+/(-	7532^*92
^	+/(-^	7532^*92
2	+/(-^	7532^*922
)	+/	7532^*922^-
+	+	7532^*922^-/
6	+	7532^*922^-/+
*	+*	7532^*922^-/+6
4	+*	7532^*922^-/+6
	empty	7532^*922^-/+64*+

Now,

7532^*922^-/+64*+

Symbol Scanned	Stack	Operation (BOPA)
7	7	
5	7, 5	
3	7, 5, 3	
2	7, 5, 3, 2	
^	7, 5, 9	A=2, B=3 ($3^2 = 9$)
*	7, 45	A=9, B=5 (5*9=45)
9	7, 45, 9	
2	7, 45, 9, 2	
2	7, 45, 9, 2, 2	
^	7, 45, 9, 4	A=2, B=2 ($2^2 = 4$)
-	7, 45, 5	A=4, B=9 (9-4=5)
/	7, 9	A=5, B=45 (45/5=9)
+	16	A=9, B=7 (7+9=16)
6	16, 6	
4	16, 6, 4	
*	16, 24	A=4, B=6 (6*4=24)
+	40	A=24, B=16 (16+24=40)

Evaluation of prefix expression using stack.

To evaluate the expressions, we scan the expression from right to left. The step involved in evaluating a prefix expression are:

Step 1: If an operand is encountered, push it on stack

Step 2: If an operator 'OP' is encountered.
(a) Pop two element of stack where A is the top element and B is the next top element.
(b) Evaluate A OP B.
(c) Push the result on stack.

Step 3: The evaluated value is equal to the value at the top of stack.

Example: + - * $ 4 2 3 3 / / 8 4 + 1 1

Symbol Scanned	Stack	Operation (AOPB)
1	1	
1	1,1	
+	2	A=1, B=1 (1+1=2)
4	2,4	
8	2,4,8	
/	2,2	A=8, B=4 (8÷4=2)
/	1	A=2, B=2 (2÷2=1)
3	1,3	
3	1,3,3	

Symbol Scanned	Stack	Operation (AOPB)
2	1,3,3,2	
4	1,3,3,2,4	
$	1,3,3,16	A=4, B=2 (4^2=16)
*	1,3,48	A=16, B=3 (16*3=48)
-	1,45	A=48, B=3 (48-3=45)
+	46	A=45, B=1 (45+1=46)

(b) +A-/B$CD*E+F-/GHI where A=1, B=2, C=3, D=2, E=1, F=5, G=9, H=3, I=2

Sol^n At first set numeric values of given operands as
+1-/2$32*1+5-/932

Symbol Scanned	Stack	Operation (AOPB)
2	2	
3	2,3	
9	2,3,9	
/	2,3	A=9, B=3 (9÷3=3)
-	1	A=3, B=2 (3-2=1)
5	1,5	
+	6	A=5, B=1 (5+1=6)
1	6,1	
*	6	A=1, B=6 (1*6=6)
2	6,2	
3	6,2,3	
$	6,9	A=3, B=2 ($3^2=9$)
2	6,9,2	
/	6,0	A=2, B=9 (2÷9=0)
-	-6	A=0, B=6 (0-6=-6)
1	-6,1	A=1, B=-6 (1+(-6)=-5)
+	-5	

Infix to prefix expression conversion:

Step 1: Reverse the input string or scan from the right to left of the infix expression.

Step 2: If it is operand, add it to the output string.

Step 3: If it is closing paranthesis, push it on stack.

Step 4: If it is an operator then,
(a) If stack is empty, push operator on stack.
(b) If the top of the stack is closing paranthesis push operator on stack.
(c) If the operator scanned has same or higher priority than the top of the stack, push operator on stack else pop the operator from stack, and add it to the output string.

Step 5: If it is an opening paranthesis, pop operator from stack and add them to the output string. Until a closing parenthesis is encountered and discard the closing parenthesis.

Step 6: If there is more input go to step 2.

pop

Step 7: If there is no more input, (unstack) the remaining operator and add them to the output string.

Step 8: Reverse the output string.

E.g. (a) 4$2*3-3+8/4/(1+1)

Symbol Scanned	Stack	Output String
)	)	empty
1	)	1
+	), +	1
1	), +	11
(	empty	11+
/	/	11+
4	/	11+4
/	/, /	11+4
8	/, /	11+48
+	+	11+48//
3	+	11+48//3
-	+, -	11+48//3
3	+, -	11+48//33
*	+, -, *	11+48//33
2	+, -, *	11+48//33
$	+, -, *, $	11+48//332
4	+, -, *, $	11+48//3324
	empty	11+48//3324$
		*-+

Reverse the output string = +-*$ which is the required string for prefix expression.

Example: (b) Consider an infix expression:
A$B*C-D+E/F/(G+H) infix form

Symbol Scanned	Stack	Output string
)	)	
H	)	H
+	), +	H
G	), +	HG
(	empty	HG+
/	/	HG+
F	/	HG+F
/	/, /	HG+F
E	/, /	HG+FE
+	+	HG+FE//
D	+	HG+FE//D
-	+, -	HG+FE//D
C	+, -	HG+FE//DC
*	+, -, *	HG+FE//DC
B	+, -, *	HG+FE//DCB
$	+, -, *, $	HG+FE//DCB
A	+, -, *, $	HG+FE//DCBA
	empty	HG+FE//DCBA$*-+

Reverse the output string = +-*$ABCD/EF+GH which is the required string for prefix expression.

(c) ((A-(B+C))*D)$(E+F)

Symbol Scanned	Stack	Output string
)	)	
F	)	F
+	)+	F
E	)+	FE
(	empty	FE+
$	$	FE+
)	$,)	FE+
D	$,)	FE+D
*	$,), *	FE+D
)	$,), *,)	FE+D
)	$,), *,),)	FE+D
C	$,), *,),)	FE+DC
+	$,), *,),), +	FE+DC
B	$,), *,),), +	FE+DCB
(	$,), *)	FE+DCB+
-	$,), *), -	FE+DCB+
A	$,), *), -	FE+DCB+A
(	$,), *	FE+DCB+A-
(	$	FE+DCB+A-*
	empty	FE+DCB+A-*$

Reverse the output string = $*-A+BCD+EF which is the required string for prefix expression.

(d) (A+B*C$D)/((E+F-G)*H)$I/J

Symbol scanned	Stack	Output string
J	empty	J
/	/	J
I	/	JI
$	/$	JI
)	/$)	JI
H	/$)	JIH
*	/$)*	JIH
)	/$)*)	JIH
G	/$)*)	JIHG
-	/$)*)-	JIHG
F	/$)*)-	JIHGF
+	/$)*)-+	JIHGF
E	/$)*)-+	JIHGFE
(	/$)*	JIHGFE+-
(	/$	JIHGFE+-*
/	//	JIHGFE+-*$
)	//)	JIHGFE+-*$
D	//)	JIHGFE+-*$D
$	//)$	JIHGFE+-*$D
C	//)$	JIHGFE+-*$DC
*	//)*	JIHGFE+-*DC
B	//)*	JIHGFE+-*DCB
+	//)+	JIHGFE+-*DCB*
A	//)+	JIHGFE+-*DCB*A
(	//	JIHGFE+-*DCB*A+
	/	JIHGFE+-*DCB*A+//
	empty	JIHGFE+-*DCB*A+//

Reverse the output string = //+A*BCD*-+EFGHIJ which is the required string for prefix expression.